ERDC MSRC/PET TR/00-35

Using OpenMP and Threaded Libraries to Parallelize Scientific Applications

by

Daniel Duffy

30 August 2000

10h0022000

Work funded by the DoD High Performance Computing Modernization Program ERDC Major Shared Resource Center through

Programming Environment and Training (PET)

Supported by Contract Number: DAHC94-96-C0002
CSC Nichols

Using OpenMP and Threaded Libraries to Parallelize Scientific Applications

Daniel Duffy
Computer Sciences Corporation
U.S. Army Engineer Research and Development Center
Major Shared Resource Center (ERDC MSRC)
3909 Halls Ferry Road
Vicksburg, MS 39180-6199

September 26, 2000

Abstract

As fast as computer processor speed and main memory size increase, scientific computing continues to push the limits of high-end machines. As high performance computers become more sophisticated, so too do the tools needed to take full advantage of their unique architectures. However, scientists have little time to explore new and interesting methods of parallel programming. This article describes two ways in which many current scientific applications used in cutting edge research may be modified quickly to take better advantage of shared-memory machines. OpenMP is a directive level parallelization method that provides code developers a quick, easy, and efficient method of performing independent operations in parallel. In many cases, changes in the existing code are confined to small regions, and most importantly, portability is retained. OpenMP is applied to two physics-related problems and both algorithms and timings are shown when running the codes on different numbers of processors. Finally, the results of solving a linear system of equations using a vendor specific threaded library are discussed. Timings for different numbers of unknowns are shown along with approximate Mflop/s (millions of floating point instructions per second) rates to show the effectiveness of using such routines.

1 Introduction

As computer architectures become more sophisticated both in hardware and software, the use of massively parallel high performance computers (HPCs) in scientific research, especially chemistry and physics, has increased dramatically. From the workstation to the vector supercomputer to new classes of HPCs, computational scientists have utilized whatever hardware was available to them. As the capabilities of these machines become greater, scientific research will continue to develop and solve larger, more complex problems.

Currently, there is an emerging class of parallel HPCs that, in several ways, are ideal for many scientific applications. These new machines are based on clusters of shared-memory processors. An excellent example of this type of machine is the SGI Origin 2000, which can be further described as a cache coherent non-uniform memory access (cc-NUMA) architecture.[1] On the Origin, two central processing units (CPUs) are connected to a shared local memory space on a single node[1] (see Fig. 1 for a schematic diagram of a node). Each processor can read and write to the shared-memory on the node such that an immediate change of a value in memory is accessible to the second CPU on that node. On the Origin, this memory access does not stop for a single node, rather memory is shared across all the nodes. Still, there is a latency associated with accessing memory that is not on the local node, thereby creating a hierarchy of memory access speeds (i.e., NUMA).

[1] The next generation SGI Origin 3000 series can contain two or four CPUs per node (or brick) and can be configured up to 512 processors.

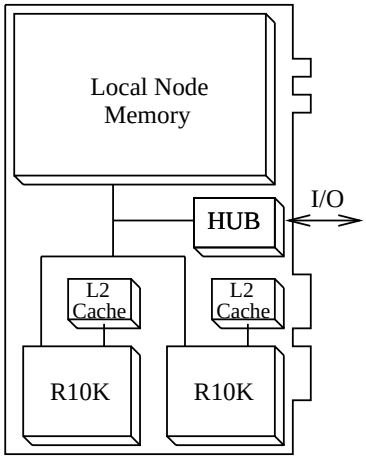


Figure 1: A simplified schematic diagram of a single node on the Origin 2000 showing the shared local memory for the two R10K processors within the node. Each processor also has its own L2 Cache and is connected to the Hub in order to perform input/output operations or communicate with other processors.

A second example is the IBM Power3 series HPC where a single node can contain up to 32 processors. Again, processors within a node have a shared-memory space, but explicit message-passing techniques must be used to share data across nodes. These distributed shared-memory hybrids lend themselves to programmatic techniques designed for both distributed and shared-memory architectures. From this, a new type of computing paradigm has emerged that uses a dual-level type of parallelism in applications.[2, 3]

In this article, the benefits of using OpenMP, a set of compiler directives and functions, to parallelize physics-type applications are discussed. An introduction to OpenMP focusing only on the loop-level *parallel do* statement is included with an emphasis on its ease of use. Example programs will be shown along with various timings and speedup analysis. Furthermore, a new framework of programming using dual-level parallelization is introduced and applied to a physics code used to calculate quasiparticle scattering rates. Finally, a short discussion on some issues of using threads and threaded libraries is included.

The machine used for the timings reported is the Origin 2000 at the U.S. Army Engineer Research and Development Center Major Shared Resource Center (ERDC MSRC), part of the DoD High Performance Computing Modernization Program.[2] This machine contains the MIPS 195 MHz R10K chip capable of two floating point operations per cycle giving a theoretical peak of 390 Mflop/s (millions of floating point operations per second). The programming language used in the examples is Fortran 77. In general, OpenMP is applicable to Fortran 90, C, and even C++ on some platforms.

2 Introduction to OpenMP

OpenMP is a collection of compiler directives and library routines that, together with the setting of environment variables, provide the programmer with an easy interface to the benefits of shared-memory parallelism. Under the OpenMP Fortran Application Program Interface (API),[4] a specification of standards has been developed to which every implementation of OpenMP should comply.[3] Hence, any existing portable code written to include OpenMP directives will retain its portability across architectures.

OpenMP compiler directives are structured as Fortran comments, written as c$OMP or !$OMP, or preprocessor directives in C, written with the #pragma OMP. These directives, of course, are only included in the compilation when the appropriate compile line flags are set. Thus, the same code may be used with and without OpenMP; a capability that may be important, for example, when sharing codes with collaborators

[2] For more information about ERDC, visit the Web site at http://www.wes.hpc.mil, and for more information about the High Performance Computing Modernization Program go to http://www.hpcmo.hpc.mil.

[3] Of course, one must make sure that their OpenMP application is compliant to the standards. When in doubt, ask your system administrator. If it is not fully compliant, make sure you understand what it is doing before using it.

```
c$OMP parallel do [clause[[,] clause]...]
        do loop
           block of statements
        end do
c$OMP end parallel do

Common clauses:
      private(variable list)
      shared(variable list)
      reduction({operator|intrinsic}:list)
      schedule(type[,chunk])
```

Figure 2: Parallel do construct with common clauses or options.

who may not be able to use OpenMP or when running on a distributed memory architecture.[4]

An OpenMP program starts out as a single process, called the master thread. When a parallel region is encountered, OpenMP forks light weight processes called threads to carry out the execution of a set of statements enclosed within a parallel construct. On shared-memory machines, the threads are placed on their own processor and parallel execution of the code takes place.[5] At the end of the parallel extent, the threads synchronize and only the master thread remains to continue execution of the program.

It is possible to use OpenMP directives in many places within a single code, thereby creating a situation where threads may be forked and joined many times throughout execution. Depending on the application and machine, this may be an undesirable situation. If the time cost of creating and destroying threads is comparable to the time of execution of the statements within the parallel region, linear speedup will not be obtained. On many machines, there are environment variables that will allow the user to specify whether threads are destroyed or kept in a sleep state waiting for the next parallel region.

Perhaps the most common and effective statement in OpenMP is the loop-level parallel do construct. Its usage along with several of its most common options are shown in Fig. 2. In order to help explain the usage of this directive and its options, a classic algorithm to compute π from the integral

$$\frac{\pi}{4} = \int_0^1 \frac{dx}{1+x^2} \tag{1}$$

is shown in Fig. 3(a). A loop over the user-specified number of iterations, `max_steps`, varies `x` from 0 to 1 while summing up the integrand. Finally, the integrand is multiplied by a factor of 4 to obtain a final value for π.

Figure 3(b) shows the same code fragment used to compute π with the inclusion of OpenMP directives to perform this loop in parallel. When compiled with the OpenMP options turned on, a set of threads will be created to perform the work done inside the loop. For this particular case, each iteration is independent making it a rather obvious candidate for parallelization.

Three clauses for the parallel do directive have been included in this example. Note that the entire OpenMP parallel do statement only affects the Fortran do loop immediately following. Furthermore, the statement has been broken up across lines using the usual Fortran 77 convention of a continuation character in the 6th column. Within Fortran 90, the prefix would be replaced with !$OMP and the usual syntax for line continuation would apply.

The *private* clause declares the list of variables considered to be local or private to a thread within the parallel extent. For the algorithm shown in Fig. 3(b), each light-weight process would then create a local variable named `i` and `x` than can only be accessed by the thread that owns that stack space. Loop variables

[4] One is tempted to include debugging as a good reason to be able to turn OpenMP on and off. However, an OpenMP application with only one thread should be the same as the original code. Only when two or more threads are used do the strange errors caused by parallel codes creep into play.

[5] Don't always assume that the thread will be bound to a unique processor. You may need to convince yourself that this is happening by running some tests on the machine you are planning to use.

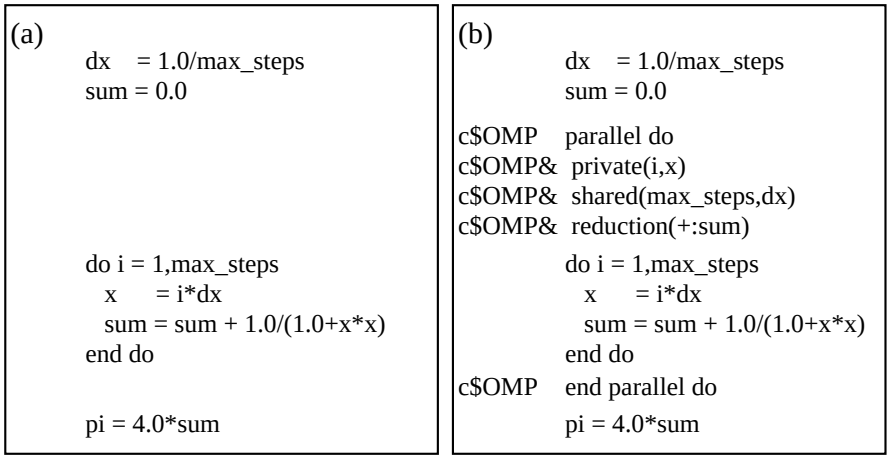


Figure 3: (a) Algorithm to compute π from Eq. 1. (b) The same algorithm with OpenMP directives included.

must be private and need not appear in any declaration statement of the parallel extent; even so, it is always good practice to explicitly declare loop variables in a private clause.

Complimentary to the private clause, a list of variables to be *shared* can also be specified. Changes to shared variables are reflected across, and immediately accessible by, all the threads working on a single problem. By default, variables are declared to be shared.

The next clause is a bit more complicated. Since the integrand within a thread and across threads needs to be summed up, OpenMP has a reduction clause that automates this for the user. A private copy of the variable `sum` is created for each thread and initialized to zero. After all the threads have finished, all the private sums along with the global sum are then reduced using whatever operation defined within the reduction clause (in this case it is a "+"). A variety of other operators can be used in a reduction statement to create a product or find the minimum or maximum of a set of numbers for example.

Table 1 shows the execution times for various numbers of threads for `max_steps` $= 10^8$. An almost linear speedup is seen as the number of processors increase from 1 up to 8. Although this may be a trivial example, it shows the power and ease of use of OpenMP directives.

Within a parallel do loop, each thread does a specific number of loop iterations independently of one another and no iteration is duplicated. OpenMP must therefore have some way to parcel out, or *schedule*, which iterations are performed on each thread. Thus, the schedule clause is used to specify how the loop iterations are to be divided between threads. In the above example, OpenMP distributed the loop iterations automatically using the default schedule type. To further explain the different types of schedulings, assume that `max_steps` $= 1000$ and that 4 threads are used. Then, the default schedule, called *static*, simply breaks up the 1000 iterations evenly between threads so that each thread gets 250 iterations, i.e., thread 0 performs i $= 1 \ldots 250$, thread 1 performs i $= 251 \ldots 500$, etc.

As shown in the usage (see Fig. 2), a *chunk* size can be specified by the user. A chunk is just the number of iterations to be given out to each thread; the chunk size in our example with 1000 iterations and 4 threads is 250. Suppose a static schedule is used with a chunk size of only 100; what iterations are performed on each of the 4 threads? For this case, thread 0 would perform iterations 1..100, 501..500, and 801..900. Meanwhile, thread 3 would only perform iterations 301..400 and 701..800. Hence, the load across all the threads would not be well balanced. The issue of a good load balance across all threads is extremely important since the resulting code is only as fast as its slowest thread.

Another type of scheduling that can be extremely useful when the load balance is not known in advance is *dynamic*. For this case, a user-specified chunk size is given out to all the threads at first. Whichever thread gets done with its iterations first immediately gets the chunk of iterations. Therefore, if the first few iterations of a loop take longer than the rest, a better load balance might be obtained using dynamic scheduling. One other type of scheduling is *guided*, which is much like the dynamic; but in this case, the chunk size can also change dynamically in order to better obtain load balancing.

In this section, the focus has been only on one user-supplied directive within OpenMP, namely the parallel

do directive. Many other capabilities exist and should be explored by the programmer to see if benefits might be found in other parts of the code other than just at the loop level. It is left up to the programmer to check for data dependencies, deadlocks, or other common bugs that occur in parallel environments. Some vendors have supplied compilers that will automatically include OpenMP directives into a user's code. Automatically including directives while checking for data dependencies is a nontrivial problem, and to date, automatic parallelization is much less sophisticated than most anything the developer of a code would include.[6]

3 Dual-Level Parallelism

Currently, many applications have already been parallelized at some coarse level using Message Passing Interface (MPI).[6] Further parallelization at some finer level would require a large overhaul and significant time coding and debugging. More than likely though, computationally intense areas of the code still exist to which OpenMP could be applied to further take advantage of the shared-memory architecture. This *dual-level parallelism* method has been applied with much success to existing MPI codes to obtain substantial increases in both resolution and turn around time without a large amount of code modification.[2] In this section, this dual-level parallel method is applied to an existing physics code, and some wall clock times are shown to demonstrate the effectiveness of the method.

The problem being solved is the calculation of the quasiparticle (q.p.) lifetimes for both s- and d-wave gaps.[7] Using a Hubbard model on a two-dimensional lattice with onsite repulsion and multiple hopping amplitudes, the spin-fluctuation interaction is calculated within a random phase approximation.[8] The calculation of the susceptibility involves a two-dimensional integral over the Brillion zone. Due to the interest in the low temperature results, finite size lattice effects become very important. Typically, lattice sizes on the order of 128×128 and 256×256 are needed to avoid any such finite size problems.

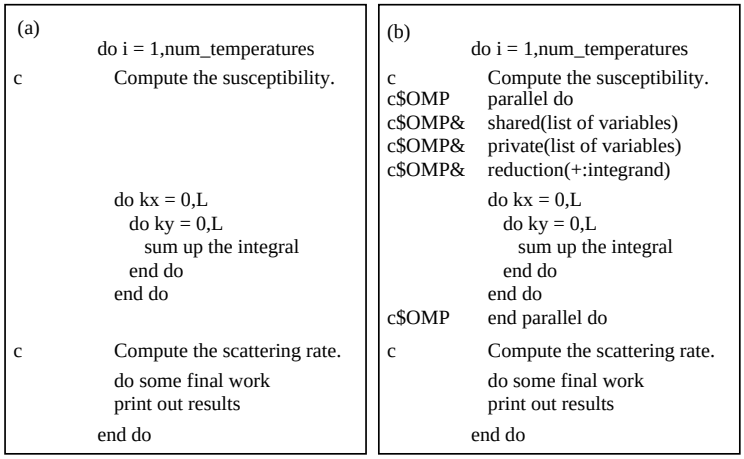
A simplified algorithm for the q.p. code is shown in Fig. 4(a) where `num_temperatures` is the number of independent temperatures and `L` is the lattice size. Since each temperature can be computed independently of the others, it is straightforward to use this as the coarse level of parallelism and implement a distributed program using MPI.

However, even with the use of symmetries that reduces the total number of steps in computing the susceptibility (i.e., the loop over `kx` and `ky`), the calculation of the integral is computationally intensive. Noting that for each value of `kx` and `ky` the integrand is independent, OpenMP directives were applied to the outer `kx` loop to implement the dual-level parallel method, see Fig. 4(b). The total source code modification took a very short amount of time (less than an hour) with the most difficult task being the correct declaration of private and shared variables.

[6] Users should not ignore the automatic parallelization capabilities of their compilers. Just do not be surprised if there is little or no speedup.

Table 1: A comparison of the execution times to compute π using the algorithm found in Fig. 3 with `max_steps` $= 10^8$. NT is the number of threads used, and all the times are in seconds. The speedup is the ratio of the time for a single processor to the time for NT processors.

NT	Time (secs)	Speedup
1	107.85	1.00
2	53.90	2.00
3	37.09	2.91
4	26.98	3.99
5	21.71	4.97
6	18.84	5.72
7	15.83	6.81
8	13.52	7.98



(a)

```
      do i = 1,num_temperatures
c       Compute the susceptibility.

        do kx = 0,L
          do ky = 0,L
            sum up the integral
          end do
        end do

c       Compute the scattering rate.
        do some final work
        print out results
      end do
```

(b)

```
      do i = 1,num_temperatures
c       Compute the susceptibility.
c$OMP   parallel do
c$OMP&  shared(list of variables)
c$OMP&  private(list of variables)
c$OMP&  reduction(+:integrand)
        do kx = 0,L
          do ky = 0,L
            sum up the integral
          end do
        end do
c$OMP   end parallel do
c       Compute the scattering rate.
        do some final work
        print out results
      end do
```

Figure 4: (a) Algorithm to compute the susceptibility used in the calculation of the lifetimes of quasiparticles in a random phase approximation for a superconductor. (b) The same algorithm with OpenMP directives included.

Table 2: A comparison of the execution times of a code to compute the quasiparticle scattering rate using a mixed mode of MPI and OpenMP on a 128×128 lattice. NP is the number of MPI processes used, while NT is the number of threads per MPI process. Hence, the total number of CPUs used for each calculation would be NP*NT. All times are in minutes, and the speedup with respect to the time of one CPU is shown in parentheses.

	NT		
NP	1	4	8
1	146.2 (1.0)	47.3 (3.1)	29.1 (5.0)
4	41.1 (3.6)	13.1 (4.2)	8.0 (11.3)
8	22.8 (6.4)	7.3 (20.0)	4.4 (33.1)

Table 2 shows the results of wall clock times when this method is applied to a 128×128 lattice. The times are in minutes, so a single-processor job running 24 independent temperatures took 2.44 hours. Meanwhile, breaking the job up into 8 independent MPI processes, each computing 3 independent temperatures, took about 23 minutes. By spawning threads, the wall clock time decreases to only 4.4 minutes using a total of 64 CPUs.

The speedup is not entirely linear, nor should that be expected. As with all parallel programs, any piece of the code that must be run in serial will eventually dominate the execution time and become a bottleneck. Since only the loop that computes the integral for the susceptibility is parallelized using OpenMP, an ideal speedup is not expected.[9] However, since the total code modification from the MPI code to the mixed MPI plus OpenMP code was only about 10 source lines taking less than an hour to implement and debug, the benefits are obvious. The turn around time has gone from over two hours to a few minutes while maintaining the portability of the code.[7]

[7] This is assuming you have access to a machine in which you can immediately submit and have your job running. Like most researchers, the amount of time spent in the queue is an important consideration. A 64-processor job is going to sit in the queue substantially longer than a 16-processor job. Hence, the users of the code must make their best judgement about the number of MPI process and threads to throw at a single problem.

4 Other OpenMP Capabilities

Through the setting of environment variables, namely `OMP_NUM_THREADS` on the Origin 2000, and by using predefined functions within OpenMP, the number of threads can be set before a job is run without having to recompile. Some implementations of OpenMP even allow the dynamic setting of the number of threads within a code, and hence the number of threads can vary during execution based on, for example, the amount of computational work. Furthermore, OpenMP includes routines to allow code developers access to the number of threads and their unique identifier. Since every thread within a single MPI process has the same MPI identifier, knowing the thread identifier is quite important.

OpenMP allows much more than just loop level parallelism; in fact, the *parallel do* construct is just a special case of the more general *parallel* construct. By creating a parallel extent within a program, OpenMP has several features to control the flow of execution of code to threads. For example, a *sections* directive defines blocks of statements to be executed by different threads. This type of situation might occur when tracking particles where calculations in the $\hat{x}$, $\hat{y}$, or $\hat{z}$ direction are independent of the others and can be performed simultaneously.

Many times throughout a piece of code, specific statements should be executed by only a single thread at a time. OpenMP allows blocks of the code to be executed either by a *single* thread or by the *master* thread. For example, such a section of code may be used to print out data to a file or increment some counter of events.

Finally, as with any parallel program, synchronization is extremely important. At the end of many parallel directives in OpenMP, an implied barrier is mandated. However, this can be controlled by the programmer using a *nowait* clause, or for that matter, a specific barrier may be used to ensure the synchronization of threads.

OpenMP is a rich and effective method for exploiting shared-memory machines. The well written manual found at the OpenMP Web site can be read in a short time with many of the features immediately applicable to most scientific codes.[4] With the guarantee of a standard, the inclusion of OpenMP into an application will remain portable. Further, with the extensive use of HPCs and their increasing popularity, the knowledge of parallel programming is a must for scientific computing.

5 Threaded Libraries

As a final example of the usefulness of threads, typical library routines found on most HPCs and workstations, namely the Basic Linear Algebra Subprograms (BLAS)[10] and the LINPACK library[5] used to solve linear equations and more, are discussed. These routines contain the usual linear algebra functions to perform vector-vector, vector-matrix, and matrix-matrix computations and can be downloaded from a variety of places on the Web. Furthermore, most vendors include a version of these routines in the system libraries that have been specifically tuned to the machine's hardware. Generally, without a large amount of work from the developer of a code, the speed of the vendor's implementation is the fastest. Hence, although it may require some work to find out how to link and call the appropriate routines from the BLAS and LINPACK libraries on any given machine, the benefits far outweigh the cost in time.

As an example, a set of equations given by the usual formula $Ax = B$ where A is a real $m \times m$ matrix whose size can be varied easily is solved. Using the `sgefa` LINPACK routine, the real matrix will be factored using Gaussian elimination. The factors computed will then be given to the `sgesl` LINPACK routine to solve for the set of equations and return the solution of x in the array holding B. As in benchmarking exercises with LINPACK,[11] both the time of execution for different size matrices as well as a Mflop/s will be reported for three different implementations of the BLAS and LINPACK routines.

The first includes the actual source of the Gaussian elimination and factor procedure along with any dependencies into the code. Second, the vendor-supplied routines on the Origin 2000 are used by linking with the *complib.sgimath* library. Finally, following the spirit of this article, the vendor-supplied threaded routines are employed to solve the set of equations in parallel by linking with the *complib.sgimath_mp* library. All versions are compiled using level 3 optimization.

Table 3 shows the results for two sizes of matrix on a single processor of the Origin 2000 (recall that the theoretical peak speed of this processor is 390 Mflop/s). Including the source code and compiling with -O3 optimization results in a Mflop/s rate that is about 6% of peak, whereas the vendor-supplied highly

Table 3: A comparison of the execution times to solve a general, real linear system of equations using both source code for the BLAS routines included into the code and the proprietary BLAS library on the Origin 2000. Times, in seconds, are shown for both a 1000×1000 and 5000×5000 matrix along with the Mflop/s rate.

	Included Source		Vendor Library	
Matrix Size	time	Mflop/s	time	Mflop/s
1,000	27.7	24.1	2.0	327.8
5,000	4900	17.0	267.4	310.1

optimized routine results in a Mflop/s rate around 80% of peak. The benefit of using the vendor-supplied libraries is extremely good for this particular set of routines, especially for the larger size matrix. Using the vendor-supplied routines decreased the run time from 1.36 hours to about 4.5 minutes for the 5000×5000 matrix. Other library routines may not perform as well, but they are definitely worth learning about and using.

Table 4 shows the results for the threaded libraries supplied by the vendor. The large linear system that required approximately 4.5 minutes on a single processor can now be solved in 35 seconds using eight processors. More than likely, the vendors are not using OpenMP to thread their routines, but the results are so dramatic that a discussion of these issues is not out of place. Note that the initial speedup in this case is almost linear but tends to fall off as the number of processors is increased. Given a specified problem size, this is to be expected; the size of the matrix to which each processor is supplied gets smaller as the number of processors increases.

In general, code that contains vendor specific routines is not immediately portable between platforms. Moving a code between different HPCs is in many cases nontrivial, and the same routines used to solve a problem may not only be named differently but their calling sequence may also change. Even so, the results shown above should be encouraging enough even for casual programmers to consider the use of vendor specific routines.

6 Conclusions

In conclusion, the use of threads either via OpenMP or from vendor-supplied libraries is a worthwhile undertaking for anyone doing computational research. OpenMP allows the developers of the code an easy and quick method of benefiting from the shared-memory architectures that are becoming more popular. With minimal effort both in learning about OpenMP and coding into an application, more processors can

Table 4: Using the vendor-supplied parallel BLAS routines, the following execution times and Mflop/s where found on the Origin 2000 for a 5000×5000 matrix using a different number of threads, NT. The final column shows the Mflop/s rate per thread or in this case per CPU. All times are in seconds, and the speedup is the ratio of time for one process to the time for multiple processes.

NT	time (speedup)	Total Mflop/s	Mflop/s/CPU
1	262.3 (1.00)	315.7	315.7
2	124.5 (2.11)	666.0	333.0
3	86.8 (3.00)	954.6	318.2
4	66.1 (3.97)	1253.0	313.2
5	53.2 (4.93)	1555.0	311.0
6	46.1 (5.69)	1792.0	298.7
7	41.9 (6.26)	1965.0	280.7
8	35.0 (7.49)	2348.0	293.5

be used to attack the problem and speed up the turn around time. Furthermore, the code, either serial or MPI, maintains its portability across architectures.

Compilers are becoming more sophisticated all the time as vendors see the need to focus their attention on such issues as OpenMP and automatic parallelization. Further, tools are being developed commercially that move beyond compilers into assurance testers for data dependencies and parallel bottlenecks and deadlocks.[12]

In many cases, threads do not result in a straight ideal (linear) speedup for a variety of reasons, some of which have already been mentioned. Making data local to a process is an extremely important factor in whether or not linear speedup is obtained. A study of cache misses may reveal much about the locality of data within a thread. Regardless of all the pitfalls that are inherent in parallel programming environments, OpenMP gives scientists a way to parallelize their codes without taking too much time away from the really important issues: science.

Acknowledgments

The author would like to thank the following people for their useful conversations and suggestions: M.R. Fahey, R. Fahey, T. Oppe, W. Ward, N. Prewitt, W. Mastin, and C. Cuicchi. This work was supported in part by a grant of computer time from the Department of Defense High Performance Computing Modernization Program at the ERDC MSRC, Vicksburg, MS.

References

[1] K. Dowd and C. Severance, *High Performance Computing*, O'Reilly and Associates: Sebastopol, CA (1998).

[2] S. Bova, C. Breshears, R. Eigenmann, H. Gabb, G. Gaertner, B. Kuhn, B. Magro, S. Salvini, and V. Vatsa, "Combining message-passing and directives in parallel applications," SIAM News **32** (1999).

[3] S.W. Bova, C.P. Breshears, C.E. Cuicchi, Z. Demirbilek, and H.A. Gabb, "Dual-Level Parallel Analysis of Harbor Wave Response Using MPI and OpenMP," The International Journal of High Performance Computing Applications, **14**, 49 (Spring, 2000).

[4] OpenMP Fortran Application Program Interface, Version 1.0, OpenMP Architecture Review Board (1997). Available online at http://www.openmp.org.

[5] E. Anderson, Z. Bai, C. Bischof, S. Blackford, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, and D. Sorensen, *LAPACK Users' Guide, Third Edition*, SIAM, Philadelphia (1999).

[6] M. Snir, S. Otto, S. Huss-Lederman, D. Walker, and J. Dongarra. *MPI-The Complete Reference: Vol. 1. The MPI Core.* 2nd Ed. Cambridge: MIT Press (1998).

[7] S.M. Quinlan, D.J. Scalapino and N. Bulut, *Superconducting quasiparticle lifetimes due to spin-fluctuation scattering,* Phys. Rev. B **49**, 1470 (1994).

[8] D. Duffy, D.J. Scalapino, and P. Hirshfeld in preparation.

[9] Ian Foster, *Designing and Building Parallel Programs*, Addison-Wesley (1995).

[10] J.J. Dongarra, J. Du Croz, S. Hammarling, and R.J. Hanson, "An extended set of FORTRAN Basic Linear Algebra Subprograms," ACM Trans. Math. Soft. **14** (1988).

[11] J.J. Dongarra, "Performance of Various Computers Using Standard Linear Equations Software." An up-to-date version of this report can be found at www.netlib.org.

[12] The best commercial OpenMP package currently available is produced by Kuck and Associates. For more information, go to their Web site at www.kai.com to learn more about Assure and Guide.